DOI 10.5162/etc2016/3.3

GUI Simulator for Automated Testing of Embedded Systems

Dipl.-Inform. Andy Walter[1], Dr. rer. nat. Torsten Rupp[2], Dipl.-Inform. (FH) Florian Weber[1]
[1] macio GmbH, Emmy-Noether-Straße 17, 76131 Karlsruhe/Germany
[2] macio GmbH, Am Kiel-Kanal 1, 24106 Kiel/Germany

Abstract:

Software Developers for embedded systems are often confronted with the situation that development tools are less mature than their pendants for desktop systems. During the development process in general, and in particular for automated testing, it can be very helpful to do frequent test-runs on the host rather than on the embedded system. Special care needs to be taken for the GUI and for low-level components like sensors.

This paper summarizes the experiences macio made with simulators in three embedded projects. With relatively little effort, it was possible to port bare-bone GUI applications which are based on low-level graphics engines (bare-metal on a framebuffer devices or simple APIs like emWin) for Linux. The main effort was the implementation of the respective low-level driver layer for the host platform, such that the original source code used on the target could be compiled for the host platform and the resulting binaries can now be executed on the host. Consequently, the screens of the resulting simulator are pixel-identic with those on the target platform.

Another benefit of this approach is that building for a second platform generally improves the robustness of the application, because it increases the probability to trigger timing-related bugs or bugs due to side-effects which may rarely become visible on the original platform.

On the host, the required time for a build-and-run cycle is dramatically reduced compared to cross-compilation and download to the target system. In total, the simulation typically reduces the turnaround time from several minutes down to a few seconds. We measured built time improvements from 6 minutes to 19 seconds on the same host. Also debugging becomes much easier and elaborated tools like Valgrind can be used for finding memory leaks and other runtime errors.

As a benefit for sales and marketing, the simulator can be used for presenting the final application to customers or producing screen shots for the documentation or brochures.

Also, the simulator can easily be integrated in automated test runs. Embedded devices often lack the possibility to generate screen shots or the memory for collecting test data. The automated handling of loading a test-program, executing it and transmitting the test-results back to the test server often is a difficult task.

Two general strategies are possible for the integration of sensors and other components: The communication can be recorded and played back by a simulator. Tools like CANoe provide more comfort and allow the execution of scripts for the simulation of smarter and more complex components. An alternative would be to only run the user interface on the host, while the non-GUI part of the application is executed on the real embedded hardware.

The paper elaborates on the benefits and limitations of the given approaches and gives guidance on the integration of the embedded device with the simulator.

Key words: GUI, simulator, embedded device, automated testing and documentation.

Motivation

Continuous testing is a required measure to ensure that the software works as expected. It improves software quality significantly. It is particularly important for embedded systems, because software failures are usually inacceptable and in some cases disastrous. Some error classes, like memory leaks or performance bottle necks, are much more likely to cause problems on an embedded system than they would on a desktop system. Embedded systems are generally only equipped with the resources which they need for the particular application, and they often run for months or even years without a restart. A memory leak in a desktop application may remain undetected, because the OS can swap and the whole application is likely to be restarted after a couple of days or weeks anyway.

Makers of operating systems and tools for embedded systems typically rather focus on things like footprint and on real-time capabilities than on comfort for software-developers. Also, cross compilation and remote debugging are more difficult to handle and have limitations.

While testing is particularly important for embedded applications, it is often difficult to run the tests on the target device. The limited resources on the target device may prevent the use of remote debuggers or intensive logging which would be comfortable. Standard concepts like the automated comparison of screen shots with a “golden master” are difficult, if there is no comfortable way to communicate with the embedded system and no space on the target device to store those screen shots.

Testing the application on a desktop PC would be more comfortable with this respect, but the target device usually runs a different operating system and even uses a different processor architecture.

Luckily, most parts of an application is usually not specific for the embedded system which is was written for. Often, only very few distinct parts really depend on the device. Especially the GUI and major parts of the business layer usually do not depend on the specific target hardware and may also run on the host system. This has the advantage that more powerful tools may be used, e.g., for code coverage analysis or runtime analyzing tools. Furthermore, testing can more easily be automated and test tools can be used.

This paper presents the experience macio made in three embedded projects with a respective GUI. A simulator framework was developed to execute the software on a standard Linux system including pixel-identic graphics.

Base and Requirements

In order to run an embedded application on a host system, the software must be compiled for the host architecture. If a make generator like cmake is used this is usually simple to do. For hand written make files, the cross-compiler must be replaced by the host standard compiler in the make files.

For the GUI part, a graphics library is needed to simulate the target graphics hardware on the host system. We used for this SDL [1] as a thin and fast graphics layer. SDL is an open-source cross-platform library with basic functions for input devices and graphics including OpenGL. For applications which use no graphics framework at all or some embedded graphics libraries like emWin, SDL is a good base to implement a simulator for the graphics backend hardware.

Concept

In order to run the embedded application with pixel-identic graphics on a host, the application must be adapted at some functional level. A good choice is the level where the graphics data is transferred to the target graphics hardware device. At this level, a system abstraction layer (SAL) is inserted in the application. Usually, this layer is very small and only contains a small number of functions. The layer is responsible for redirecting the graphics output to SDL functions, which draw the GUI on the host system screen. This low-level approach offers the ability to use most of the higher level embedded application graphics functions including testing and give a pixel-identic output on the host system.

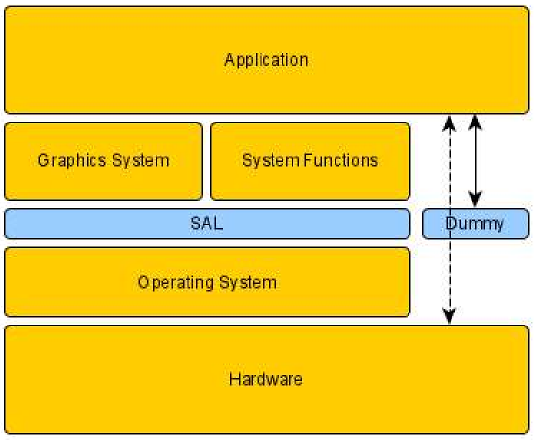


Fig. 1. Target system with application and the level were the SAL was introduced.

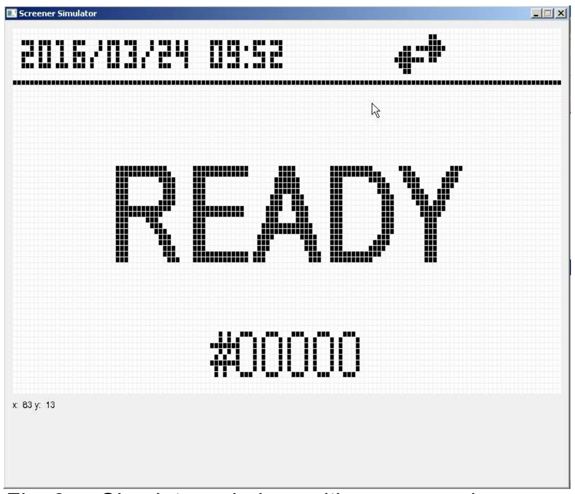
All non-graphical functions, e.g., operating system calls, are also redirected in the SAL to an appropriate host function, e.g., opening a file. If the functionality is not required for testing, it may just be implemented as a dummy. This approach results in a partially operating application. E.g., sensor communication may not work, but the correct visualisation of some recorded or otherwise given sensor data may be tested. For development and testing of the GUI and major parts of the business layer, usually none or only a few hardware dependent functions are required. Only this sub-set must be replaced.

Implementation

The approach of building the embedded application for the host system was successfully used in three different projects at macio.

1) Medical emergency respirator system: for this device a graphical user interface was implemented based on an external graphics controller connected to the main system via an SPI bus. For the simulator on the host, the low-level commands for sending and receiving data via the SPI bus were replaced in the SAL by calls to SDL functions. The graphics controller supported different screen buffers and semi-translucent overlay screen buffers, which must be implemented with SDL functions to get a pixel-identic output on the host. The hardware buttons where implemented as software controls in the SDL window, too.

2) Alcohol measurement device: in this project, the graphical user interface of an alcohol measurement handheld system with a monochrome screen of only 128x64 pixels was developed. The application uses emWin [2] for the graphics output. The SAL was inserted at the level were the rendered data from emWin was copied onto the screen buffer. The graphics data was redirected to SDL blit-operations in a window on the host system.


Screener Simulator

2016/03/24 09:52

READY

#00000

x: 83 y: 13

Fig. 2. Simulator window with an example screen of the alcohol measurement handheld application.

3) Embedded system with a complex generic GUI for a large product family on Cortex M-CPUs: this project was based on a domain-specific language, which makes the GUI easily adaptable to other products of the same family, by just replacing the GUI description. The project uses emWin on a frame buffer device for the graphical display. The SAL was inserted at the level were the rendered data from emWin was copied into the frame buffer.

Development benefits

With the major part of the target application running on a Linux host system, most of the further development work could be done one the host including debugging and runtime analysis. The time for an edit-compile-run cycle was dramatically reduced from typical several minutes to a few seconds, both because the standard compilers of the host could be used and the resulting binary could be started immediately without downloading it to the target or copying it to a memory card first. For the largest project, the edit-compile-run cycle was reduced from 6 minutes on the original Windows 7 development environment with a cross-compiler down to 19 seconds on a Linux system on the same desktop hardware.

Besides faster development cycles, runtime analysis is another major benefit when the application can run on a standard Linux x86 system. Especially dangling pointers, buffer overflows, corrupted memory, or lost resources can be tracked and easily identified with tools like Valgrind [3]. On a target system, this is often impossible due to limited system resources (memory and CPU power) or uncooperative processor architectures.

If performance is a critical factor, then a performance analysis may also be done on the host system. The results are not absolutely comparable with the real target system, but measurement values like counters for executed code blocks or functions calls can be used to find platform-independent hot-spots in the application.

Sensor integration

If sensor data is required to execute the embedded application, the data can be injected into the host application on various ways. If a field bus like CAN is used, software tools like CANoe [4] or similar tools may be used in order to simulate a CAN network. With a simulated CAN device, network development of the application can start even before the final CAN backend hardware is available. This approach is particularly convenient for testing product families, when some of the devices would be too big, too expensive or simply not available yet.

If the application has an internal structure like a publish-subscribe pattern for handling sensor data and internal application state information, it is easily possible to extend this by a simulator connector to get and inject sensor and state data from and into this model. The medical respirator system uses a publish-subscribe model and a few additional injection functions to transmit basic sensor data to the host application, too.

Automated testing and documentation

For automated testing, a socket interface was implemented in the simulator framework and the script language Lua [5] was integrated. Lua is used for writing lightweight test scripts. The Lua integration offers the ability to insert user, communication, and simulator events, and for taking and comparing screenshots of the whole screen or regions of interest. Several use cases for the various applications were implemented and the graphical output of the tests was compared with fuzzy rules with the expected screen data. Fuzzy rules were used to limit the overhead and complexity of the comparison function e. g. when text in different languages is rendered. This is sufficient for some general rendering tests. E.g., the automated detection if all text labels fit in the respective space and can be completely drawn in all supported languages.

Based on the Lua integration, a screenwalker script was implemented to be able to instantiate all application screens by an automated menu navigation walk-through. Via this screenwalker functionality, a rendering of all screens can be ensured to test all defined fuzzy rules in one automated test run. Furthermore, the screenwalker can be used to generate a screen map of all screens provided by the application.

When tests can be executed on the host, also a code coverage analysis can be done. On the target system, this is in many cases impossible unless there is a writable storage device available were the coverage analysis results may be stored. With the host simulator, standard coverage tools like GNU gcov can be used to get a detailed code coverage analysis.

Besides automated tests and coverage analysis, creating screen shots with rendered texts in all required languages for the user manual is in many projects a major requirement. Doing this manually on the real target could become a time-consuming task. By using the pixel-identic simulator with the automated screenshot functionality, those screenshots can be created and updated fully automated for all required screens and languages.

Future work and conclusions

Executing the embedded application in a simulated environment on the host is already a major benefit during software development, test and documentation. For detailed testing and for demonstration purposes, some real sensor data may be useful. Implementing simulator functions which supply realistic sensor data is in many cases a very complex task. Alternatively, real sensor data may be collected from a connected target device during the test run and then sent to the simulator software on the host via a network connection. This would also allow remote control of the embedded device for debugging as well as for demonstration purposes.

Using a simulator for the development of an embedded application is a major benefit. The time for an edit-compile-run cycle can dramatically be reduced. Also, collaborate software development with several developers is easier now if there are not enough target devices for the whole team. With a simulator, testing and generating documentation can easily be automated and a simulator is also useful for marketing purposes to give customers an impression of the real system.

Finally, the GUI simulator can be used to decrease the effort and time for text translations of multi-lingual applications. In most cases a translation agency gets authorized to translate all origin application strings into different languages. This agencies need context information about the text placement to be able to find suitable translations. Often, this context information is provided by screen maps and developer comments, which may lack of expressiveness. The GUI simulator is able to provide this context information by presenting the complete application including the navigation path and the whole screen of the translated text. Additionally, the translation agency can use the GUI simulator for verifying the translated text. The look and feel of the translated text can be displayed directly using a dynamic translator implementation. Therefore the translation agency is able to verify the correctness of text placing concerning newlines and the available space.

In the three presented projects the effort required to develop the simulators was more than compensated by the saved time due to faster development cycles and easier debugging. For future projects, the simulator framework will be used more extensively and building a simulator should become part of the project offer.

References

[1] SDL: Simple Direct media Layer, http://www.libsdl.org/

[2] emWin: https://www.segger.com/emwin.html

[3] Valgrind: http://valgrind.org/

[4] CANoe: http://vector.com/vi_canoe_de.html

[5] Lua: http://www.lua.org/